

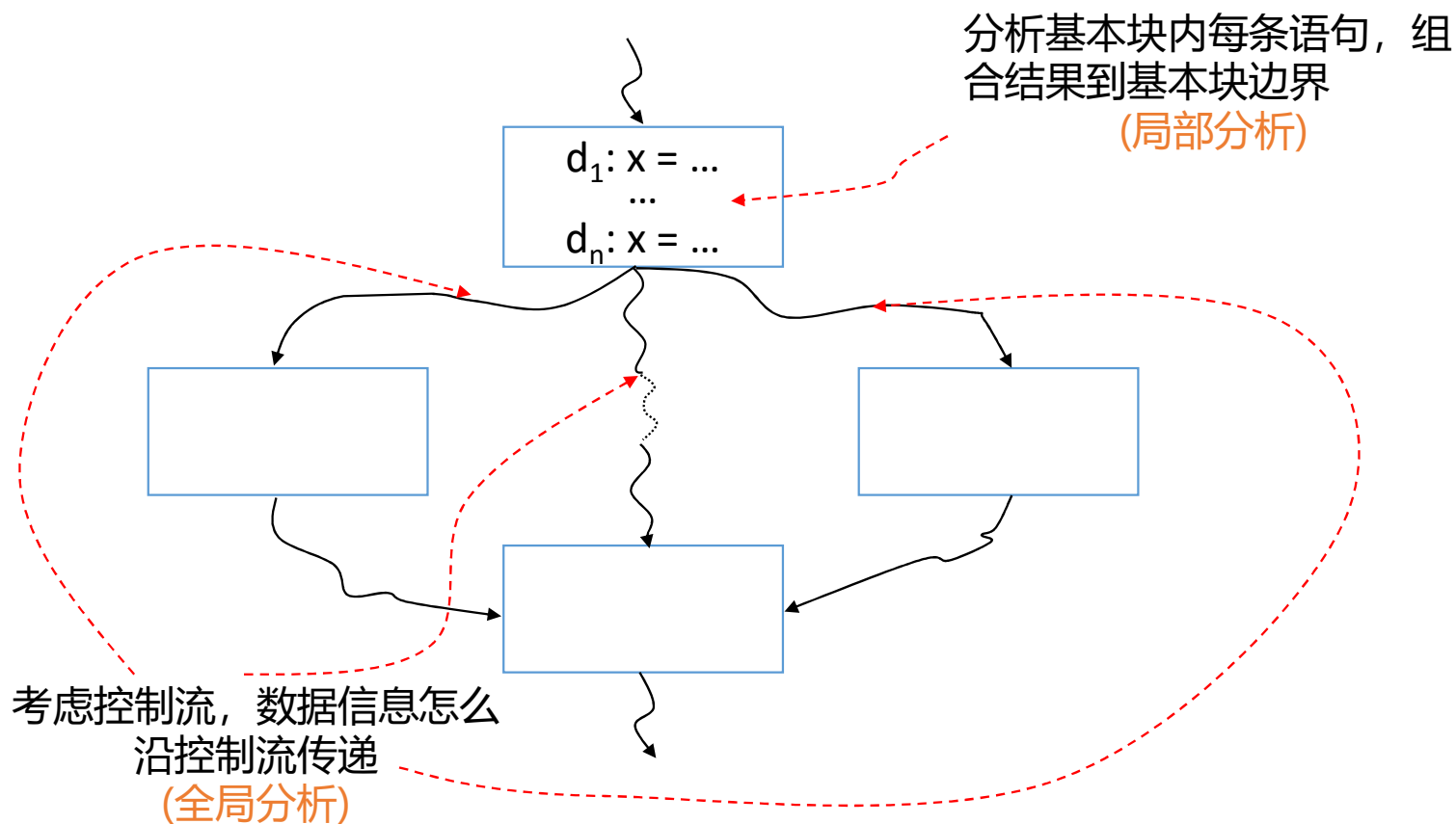
并行编译与优化 Parallel Compiler and Optimization

计算机研究所编译系统室

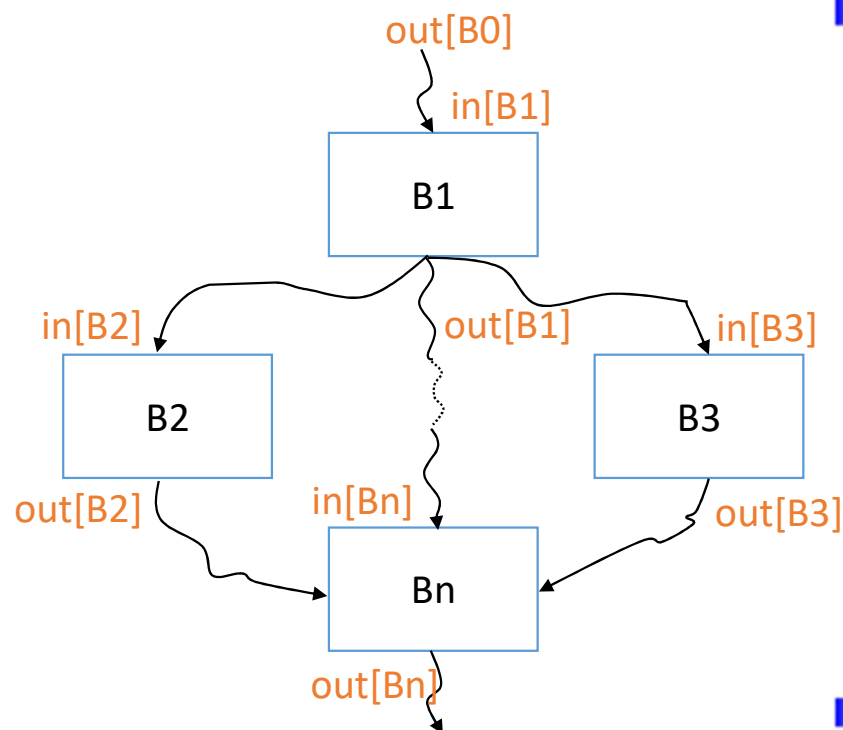
Lecture Six: Scalar Optimization

第六课：标量优化

迭代的数据流分析框架



迭代的数据流分析框架



数据流方程的建立依赖于程序语句对求解的数据问题的影响

- 建立一组方程，对于每个基本块，求解出口点数据流信息集合 $out[B]$ 和入口点数据流信息集合 $in[B]$ ：

- ⊕ 基本块内语句（代码）对求解的数据流问题的影响 转移函数

$f_B: in[B] \rightarrow out[B] / out[B] \rightarrow in[B]$

- ⊕ 控制流作用：

如果B1、B2有边连接， $out[B1]$ 和 $in[B2]$ 之间的关系

- 寻找方法求解方程组（迭代求不动点）

迭代的数据流分析框架

■ 迭代的通用数据流分析框架：

- ⊕ 一个数据流图，包含两个特殊的结点Entry和Exit
- ⊕ 数据流方向D
- ⊕ 一个值集V
- ⊕ 一个meet操作 \wedge
- ⊕ 一个传递函数的集合F， f_B 表示基本块B的传递函数
- ⊕ V的一个常量值 V_{entry} 或 V_{exit} ，分别表示前向和逆向框架的边界条件

■ 输出：每个基本块边界处V的值

4、迭代的数据流分析框架

■ 前向的数据流分析

```
out[Entry] =  $V_{\text{entry}}$ ;  
for (each  $B \in N - \{\text{Entry}\}$ )  
    out[B] =  $T$  //初值  
while(change to any out occur){  
    for(each  $B \in N - \text{Entry}$ ) {  
        in[B] =  $\bigwedge_{p \in \text{pred}[B]} \text{out}[P]$ ;  
        out[B] =  $f_B(\text{in}[B])$ ;  
    }  
}
```

4、迭代的数据流分析框架

■ 后向的数据流分析

```
in[Exit] = Vexit;  
for (each B ∈ N - {Exit})  
    in[B] = T //初值  
while(change to any in occur){  
    for(each B ∈ N - Exit){  
        out[B] =  $\bigwedge_{s \in \text{succ}[B]} \text{in}[S]$ ;  
        in[B] =  $f_B(\text{out}[B])$ ;  
    }  
}
```

冗余优化

■ 中间代码中存在大量冗余指令

- ⊕ 公用子表达式
- ⊕ 循环不变量
- ⊕ 死代码
- ⊕ 部分冗余.....

冗余优化

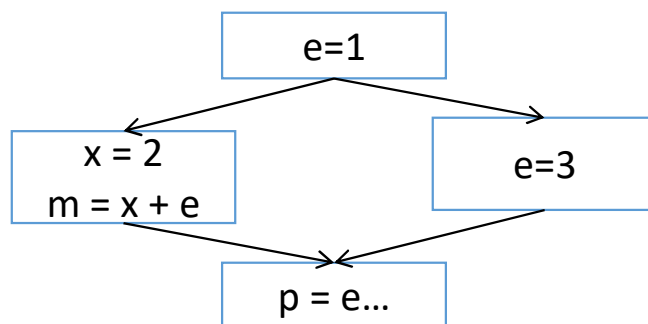
■ 一些常用的用于优化技术

- ⊕ 公用子表达式删除 (Common subexpression elimination)
- ⊕ 值编号 Value numbering (Local & Global)
- ⊕ 常数传播/复制传播 (Const propagation/Copy propagation)
- ⊕ 循环不变量外提 (Loop-invariant code motion)
- ⊕ 死代码删除 (Dead code elimination, DCE)
- ⊕ 代码提升 (Code Hoisting)
- ⊕ 部分冗余删除 (partial redundancy elimination, PRE)

内容

- 常数传播
- 公用表达式删除
- 值编号
- 循环不变量外提
- 死代码删除
- 激进的死代码删除

1、常数传播



- 常数传播在每个基本块的边界，判断每个变量是否是常数？
 - ⊕ 如果是，值是多少？
 - ⊕ 和到达定值和活跃变量分析不同，常数传播的可能的数据值的集合是无界的。
- 常数传播是一个前向数据流问题

1.1 常数传播

```
out[Entry] =  $V_{\text{entry}}$ ;  
for (each  $B \in N - \{\text{Entry}\}$ )  
    out[B] = 未定义 //初值  
while(change to any out occur){  
    for(each  $B \in N - \text{Entry}$ ) {  
        in[B] =  $\bigwedge_{p \in \text{pred}[B]} \text{out}[P]$ ;  
        out[B] =  $f_B(\text{in}[B])$ ;  
    }  
}
```

常数传播，是一个“必然”的问题， \bigwedge 是“集合交”

1.1 常数传播

- $\text{in}[S, x], \text{out}[S, x]$ 分别表示语句 S 前后变量 x 的信息。 $\text{out}[S, x] = f_s(\text{in}[S, x])$
- 边界条件有，对所有变量 x ， $\text{out}[\text{entry}, x] = \text{未定义}$ 。

1.1 常数传播

■ 按以下方式定义传递函数

- ⊕ 如果S不是赋值语句，那么 f_s 是单元函数，即 $f_s(x) = x$
- ⊕ 否则，假设 $S: x \leftarrow \text{RHS}$ ，对所有 $v \neq x$ 的变量，有：

$$\text{out}[S, v] = \text{in}[S, v]$$

① 如果RHS是常数 c ，则 $\text{out}[S, x] = c$

② 如果RHS形如 $y \otimes z$ ，则：

如果 $\text{in}[S, y]$ 和 $\text{in}[S, z]$ 都是常数，则 $\text{out}[S, x] = \text{in}[S, y] \otimes \text{in}[S, z]$

如果 $\text{in}[S, y]$ 和 $\text{in}[S, z]$ 中有一个不是常数，那么 $\text{out}[S, x] = \text{NAC}$

否则 $\text{out}[S, x] = \text{undef}$

③ 如果RHS是其他表达式，则 $\text{out}[S, x] = \perp$ //不是常数

1.2 常数传播优化

首先进行常数传播分析

对每个基本块 $B \in N$, 执行 {

对每条 $instr \in B$, $instr$ 使用的 x , 如果在 B 的入口点 x 是常数 c {

① 判断在这条指令处 x 是否为常数 c

② 替换 $instr$ 中的 x 为 c

}

}

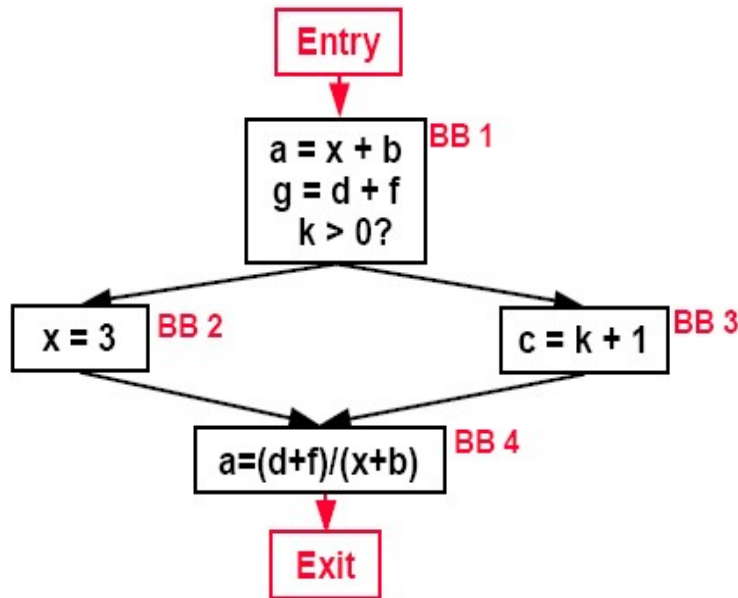
2、公用子表达式删除

- 对程序中表达式的一次出现，如果存在该表达式的另一次出现，并且执行顺序上它总是在该表达式之前计算，并且这两个计算之间表达式的操作数没有发生变换，我们就说该表达式的这次出现是公用子表达式。
- 公用子表达式删除就是用保留的结果替换重复计算。

2.1 可用表达式

- 如果一个表达式 $x \otimes y$ 在点 p 满足下面的条件，我们就说该表达式在点 p 可用：
 - ⊕ 从entry到 p 的每一条路径都计算 $x \otimes y$
 - ⊕ 在到达 p 之前最后一次计算 $x \otimes y$ 后，没有对 x 或者 y 的赋值
- 可用表达式分析就是找出每个基本块 B 的边界处所有可用的表达式。
- 可用表达式的结果用于公用子表达式的删除。

2.2 可用表达式分析



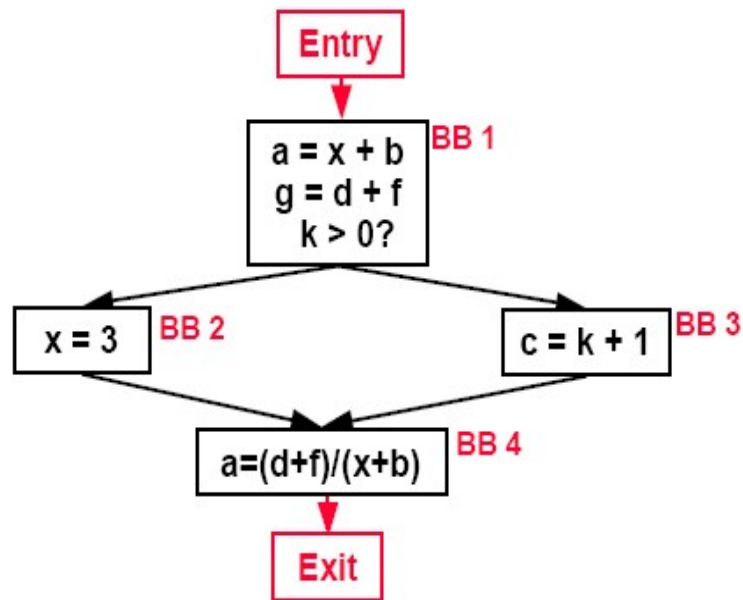
■ 可用表达式分析需要定义基本块的两个集合：

- ⊕ $eval(B)$: 在基本块中计算的、并且在基本块的出口点仍旧可用的表达式集合
- ⊕ $kill(B)$: 被基本块B杀死的表达式集合

$$in[B] = \bigcap_{p \in pred[B]} out[p]$$

$$out[B] = eval(B) \cup (in[B] - kill[B])$$

2.2 可用表达式分析

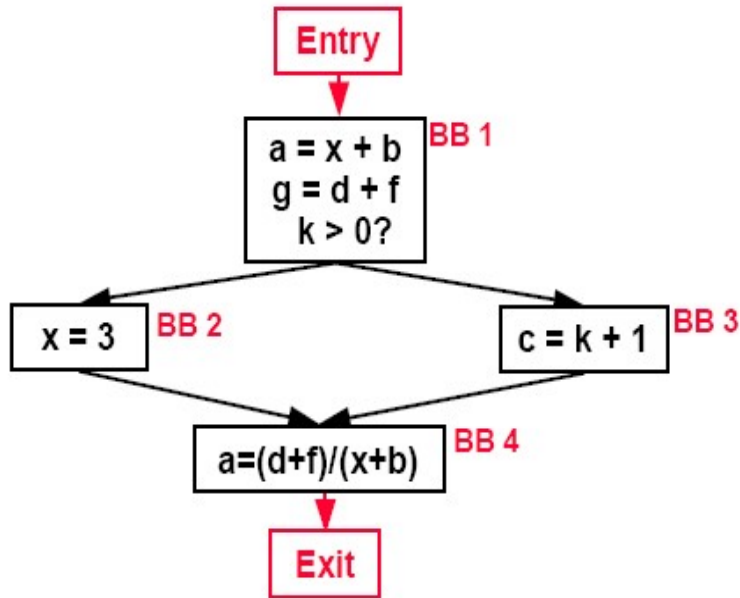


- $in[B]$: B的入口点的可用表达式的集合
- $out[B]$: B的出口点可用表达式的集合
- 可用表达式寻找的是从Start结点到点p的可用表达式，因此是一种前向数据流分析问题。

$$in[B] = \bigcap_{p \in pred[B]} out[p]$$

$$out[B] = eval(B) \cup (in[B] - kill[B])$$

2.2 可用表达式分析



- $in[B]$: B的入口点的可用表达式的集合
- $out[B]$: B的出口点可用表达式的集合
- 可用表达式寻找的是从Start结点到点p的可用表达式，因此是一种前向数据流分析问题。

$$in[B] = \bigcap_{p \in pred[B]} out[p]$$

$$out[B] = eval(B) \cup (in[B] - kill[B])$$

2.3 公用子表达式删除

首先进行可用表达式分析

对每个基本块 $B \in N$ ，执行 {

对每条 $instr \in B$ ，如果 $instr$ 形如 $z = x \otimes y$ ，且 $x \otimes y$ 在 B 的入口点可用 {

- ① 判断 $x \otimes y$ 是否在这条指令处可用
- ② 获得到达 z 的 $x \otimes y$ 计算 ($instr$ 除外)
- ③ 创建新的临时变量 t
- ④ 步骤②获得的定值指令 $w = x \otimes y$ ，替换为两条连续指令

$t = x \otimes y ; w = t$

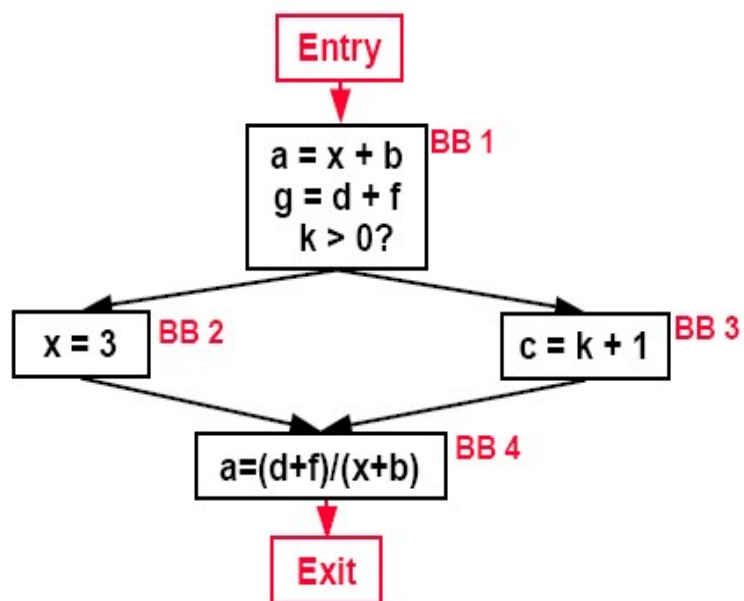
5. 将 $z = x \otimes y$ 替换为 $z = t$

}

}

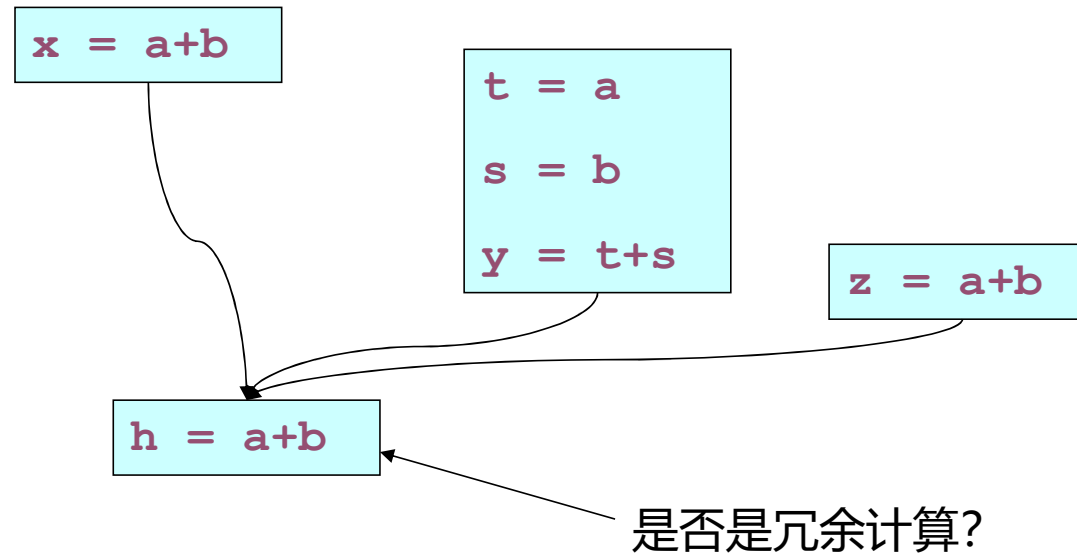
2、公用子表达式删除

2.3 公用子表达式删除



画出CSE优化后的控制流图。

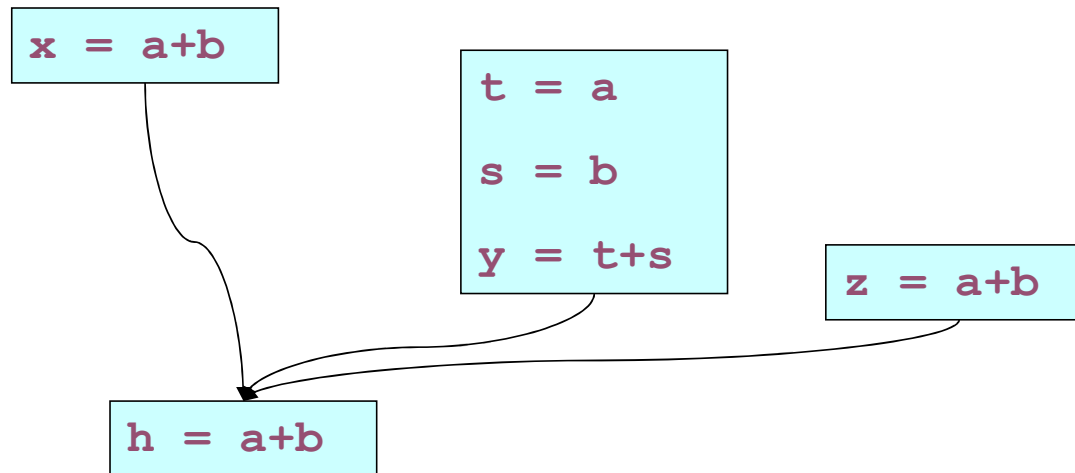
3、值编号 (Value Numbering)



■ 公用子表达式删除 (CSE) : **NO!**

■ 值编号 (VN) : YES!

3、值编号 (Value Numbering)

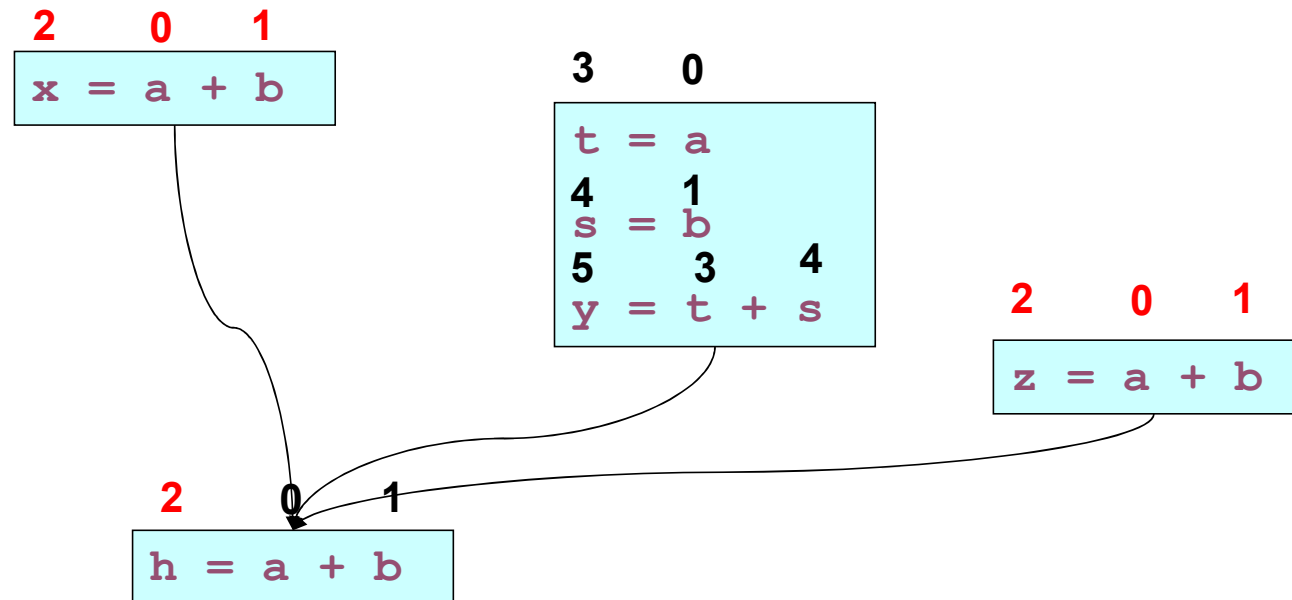


思想:

- ⊕ 每个表达式赋予唯一的符号值——编号 (Value Numer)
 - 每个变量、表达式、常数
- ⊕ 全等
 - $VN(a) = VN(b)$, 则a全等于b
 - 如果 \oplus 和 \otimes 是相同的操作符, 且x全等于a & y全等于b, 那么 $x \oplus y$ 全等于 $a \otimes b$ (考虑: \oplus 满足交换律的情况)

3 值编号 (Value Numbering)

- 每个表达式赋予唯一的值编号VN
- 全等
 - $VN(a) = VN(b)$, 则a全等于b
 - ⊕ 如果 \oplus 和 \otimes 是相同的操作符, 且x全等于a & y全等于b, 那么 $x \oplus y$ 全等于 $a \otimes b$ (考虑: \oplus 满足交换律的情况)



3.1 计算值编号

■ 编译器如何实现值编号？—— 基于hash值的值编号

⊕ $a = \text{常数 } c$

$$\text{VN}(a) = c$$

⊕ 表达式 $x = y \otimes z$

$$\text{VN}(x) = \text{hash}(\otimes, \text{VN}(y), \text{VN}(z))$$

⊕ 根据 \otimes 的可交换性选择不同的Hash函数

3、值编号

3.2 局部值编号

- 1、每条指令 $r = \langle \otimes, op1, op2 \rangle$ 时，计算hash值（值编号）
 - ① Hash查找hash($op1$)和hash($op2$)，如果hash值不在表中，加入表中
 - ② Hash(r) = Hash($\langle \otimes, op1, op2 \rangle$)
 - ③ 如果hash(r)值已存在，则用表中值对应的指令左部来代替表达式计算

$a = i+1$
 $b = 1+i$
 $i = j$
 $t1 = i+1$
if($t1$) goto L1

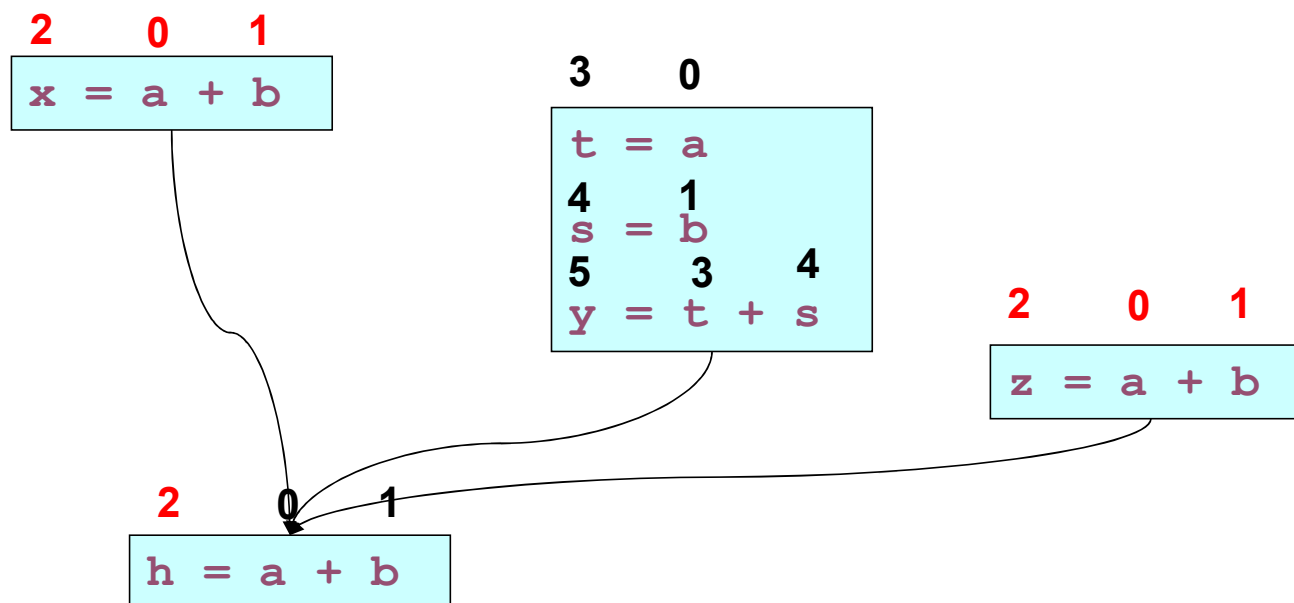
Hash	Rhs	lhs
h1	i	
1	1	
h2	i+1	a
h3	j	i
h4	i+1	t1

$a = i+1$
 $b = a$ //删除表达式1+i
 $i = j$
 $t1 = i+1$
if($t1$) goto L1

$$h4 = \text{hash}(+, h3, 1) \neq \text{hash}(+, h1, 1)$$

3、值编号

3.2 全局值编号

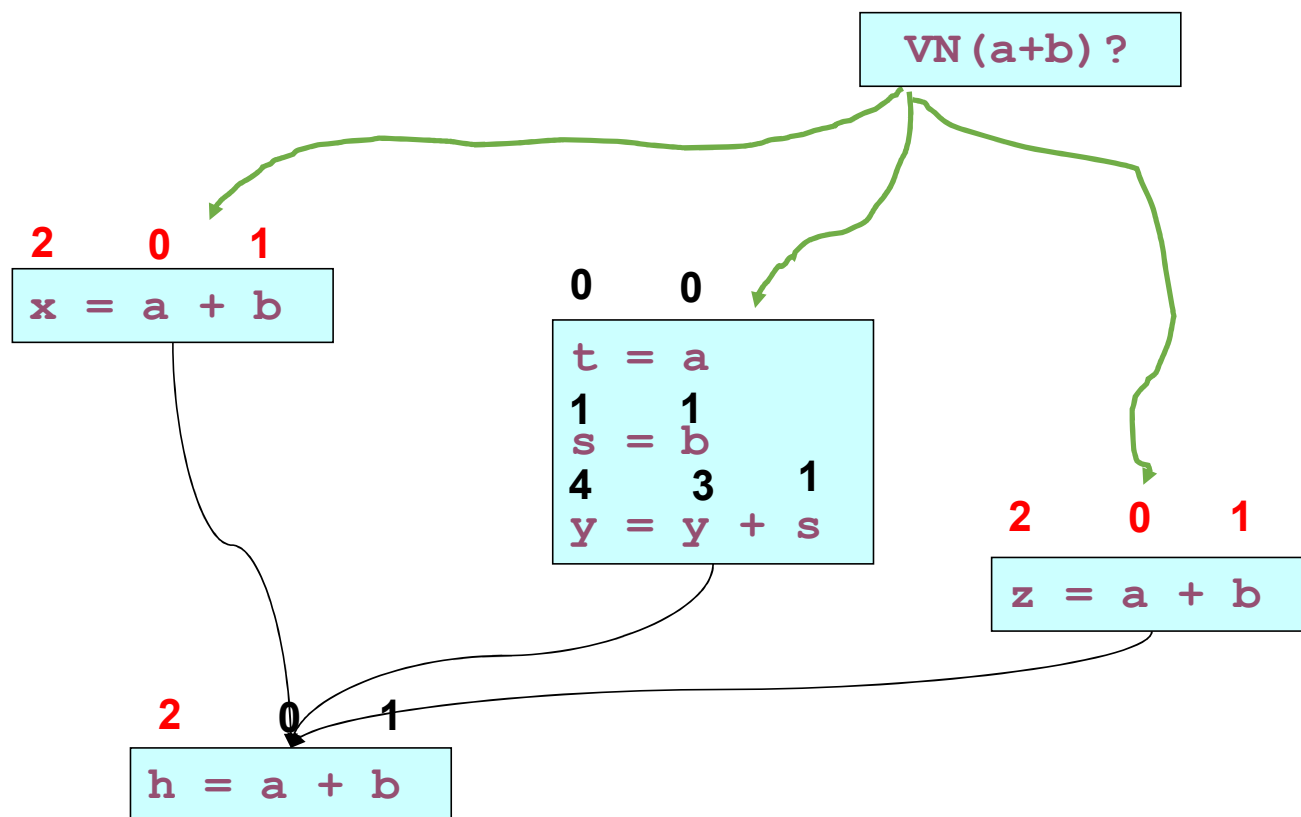


- $h = a + b$ 其中, $a + b$ 冗余表达式
⊕ 部分冗余!

3.2 全局值编号

■ 基于必经节点的全局值编号

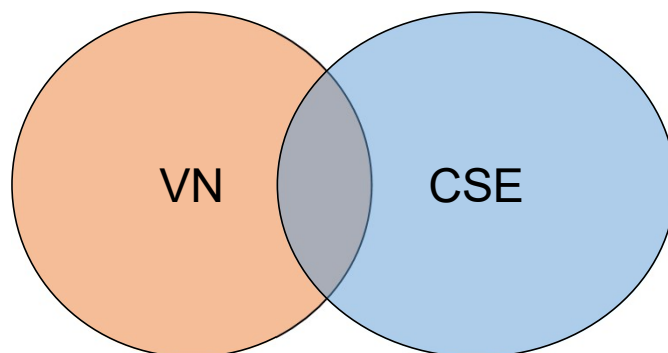
⊕ Dominator-based Global Value Numbering



- ⊕ 使用直接必经节点的hash表
- ⊕ 基于SSA形式展开

3 值编号 (Value Numbering)

■ CSE vs VN



■ 冗余删除的基本方法

⊕ CSE: 语法角度

⊕ VN: 语义角度

■ 都实现!

4 循环不变量外提

- 如果循环中的一个计算在循环的每次迭代都产生相同的值，则该计算被称作**循环不变量**。
- 当循环的基本块的一个操作的操作数都满足下列条件时，该操作是循环不变的：
 - ⊕ 操作数是常数，或者
 - ⊕ 所有到达操作数的定值都在循环之外，或者
 - ⊕ 只存在一个到达操作数的定值，并且该定值在循环内，并且该定值本身是循环不变的

4.1 循环不变量

■ 标记循环中的不变量集合

1、首先标记两种简单的不变量

- ① 标记所有操作数都是常数的指令为不变的
- ② 标记所有操作数的到达定值都在循环外的指令为不变的

2、循环直到找出了所有的循环不变量：

标记操作数符合下面条件，并且没有被标记的指令，为不变的

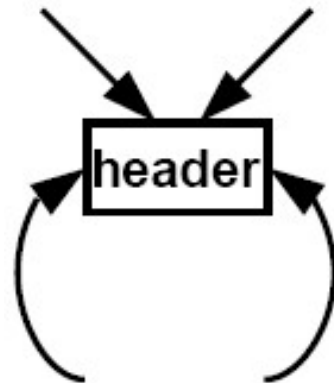
- ① 操作数是循环不变量，或者
- ② 只有一个定值到达操作数，并且该定值已经标记为循环的不变量

4.2 循环前置节点

■ 思考？

不变量外提后，放在哪里？

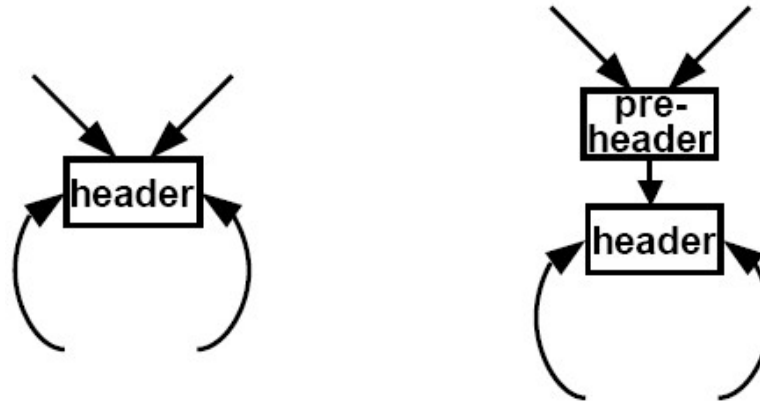
⊕ 循环头节点？ 循环头节点的前驱？



4.2 循环前置节点

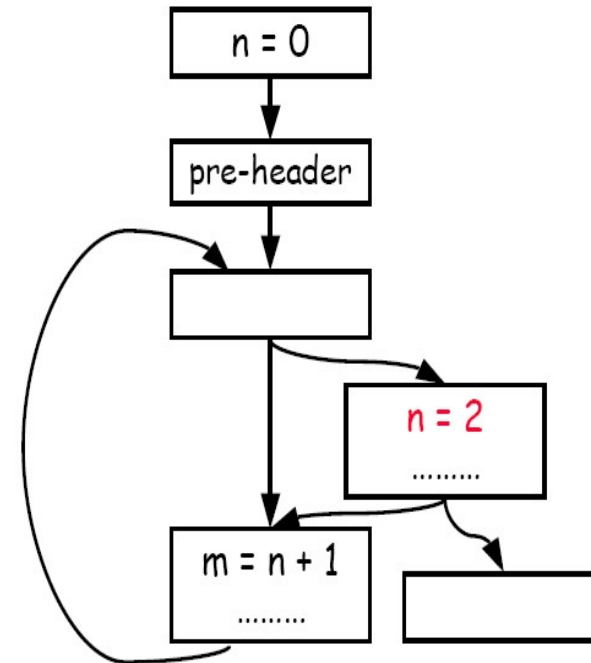
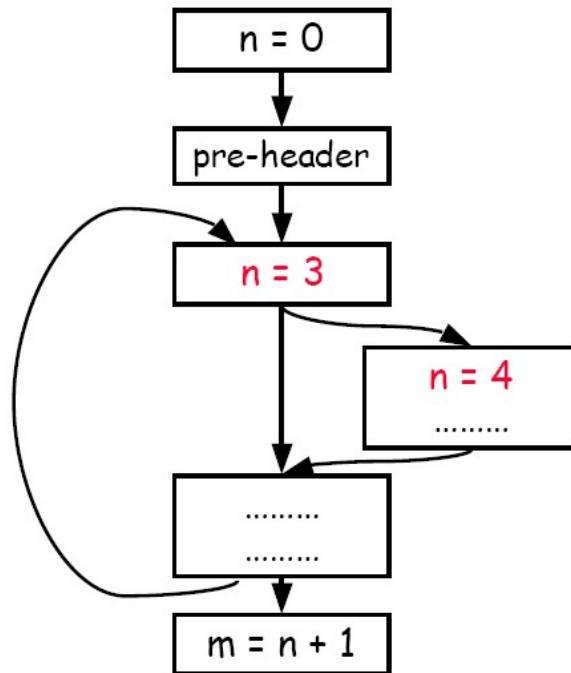
■ 循环前置结点

- ⊕ 只有一个后继：循环头结点
- ⊕ 来自于循环之外、到循环头结点的边，改为到前置结点
- ⊕ 来自于循环内部，到循环头节点的边，保持不变



4.3 循环不变量外提

- 循环不变量移动是将发现的不变量外提到循环头。
- 所有不变量都可以外提么？



4.3 循环不变量外提

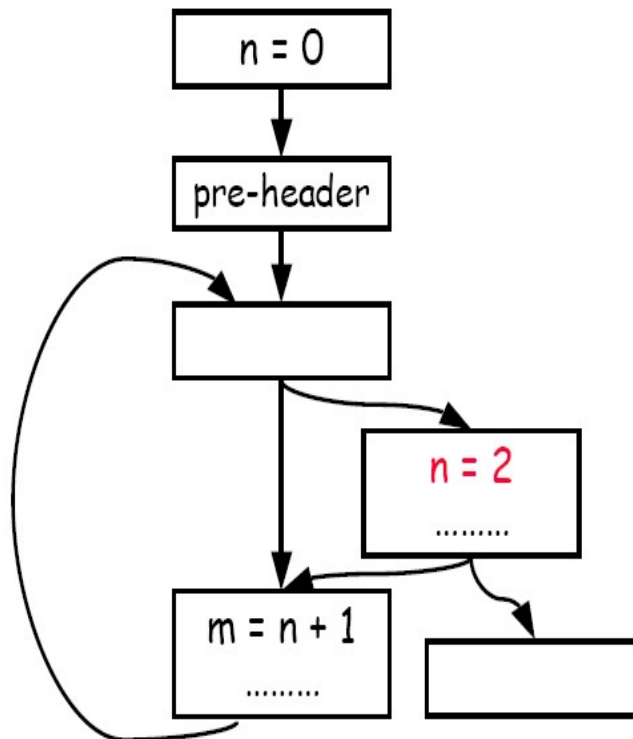
- 循环不变量移动是将发现的不变量外提到循环头。
- 循环不变量外提条件

假设不变量s: $v = x \otimes y$

对 v 进行定值, 使用 x 、 y

6.3 循环不变量外提

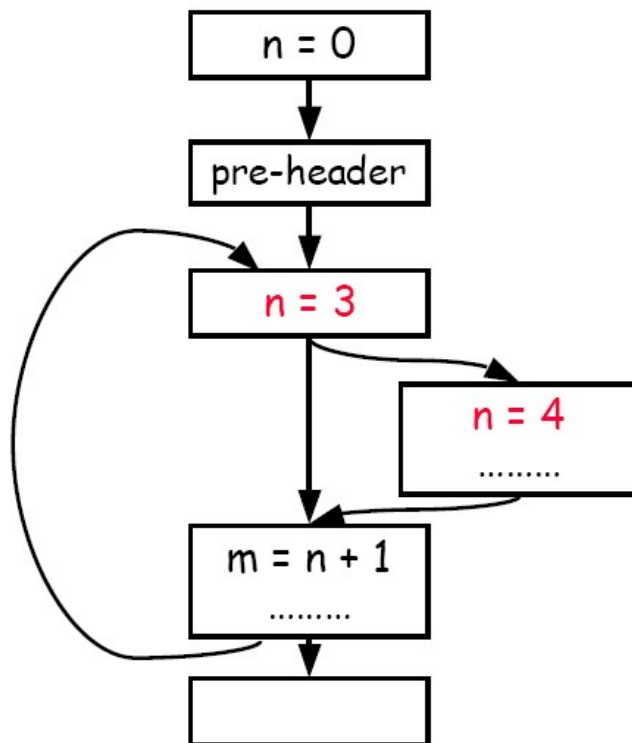
■ 不变量外提条件一



条件一：语句s是循环L所有出口的必经结点

4.3 循环不变量外提

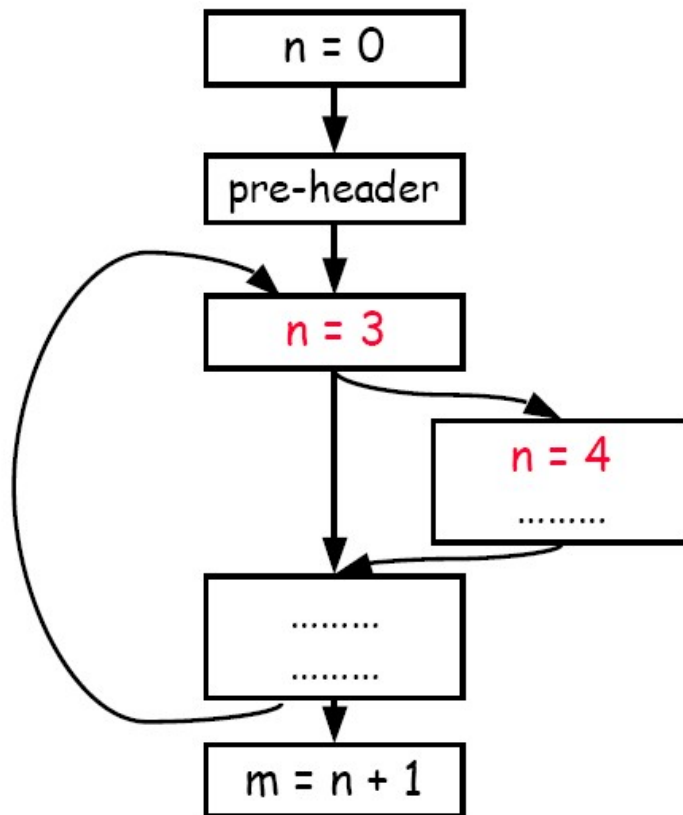
■ 不变量外提条件二



条件二：循环中对 v 的所有使用只能被 s 中的定值到达

4.3 循环不变量外提

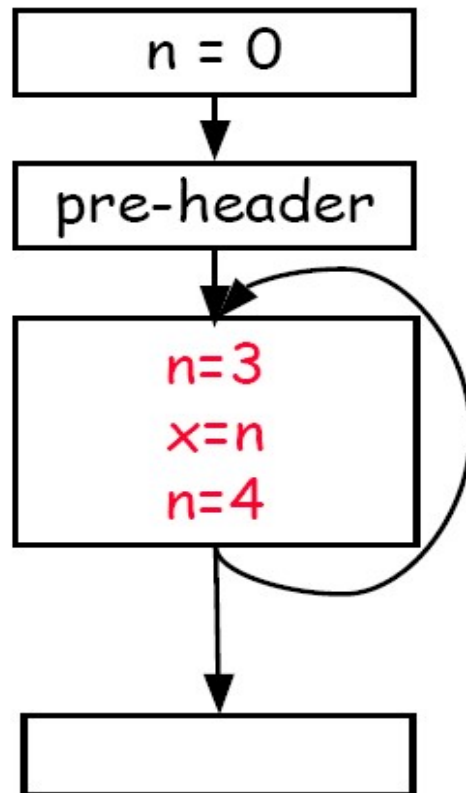
■ 不变量外提条件三



条件三: **v**不能在循环L内有其他定值

4.3 循环不变量外提

■ 不变量外提条件四



根据前三个条件，`x=n` 被外提，但是`n=3` 和`n=4`不能外提，显然发生了错，因此限制条件四：考虑不变量使用的操作数.

条件四：不变量的使用的`x`、`y`的定值都来自于循环之外

4.3 循环不变量外提

执行到达定值分析，并构建UD链

找出循环L中的循环不变量

对每个循环不变量指令s，假设s定值v，使用x和y，执行{

如果 (1. s所在基本块是L的所有出口的必经结点 &&

2.在循环L中v没有其他定值 &&

3.并且L中v的所有使用只被s中的定值到达 &&

4. x和y的定值在循环外 (可以是一开始就在循环外，或者外提后在循环外)

{

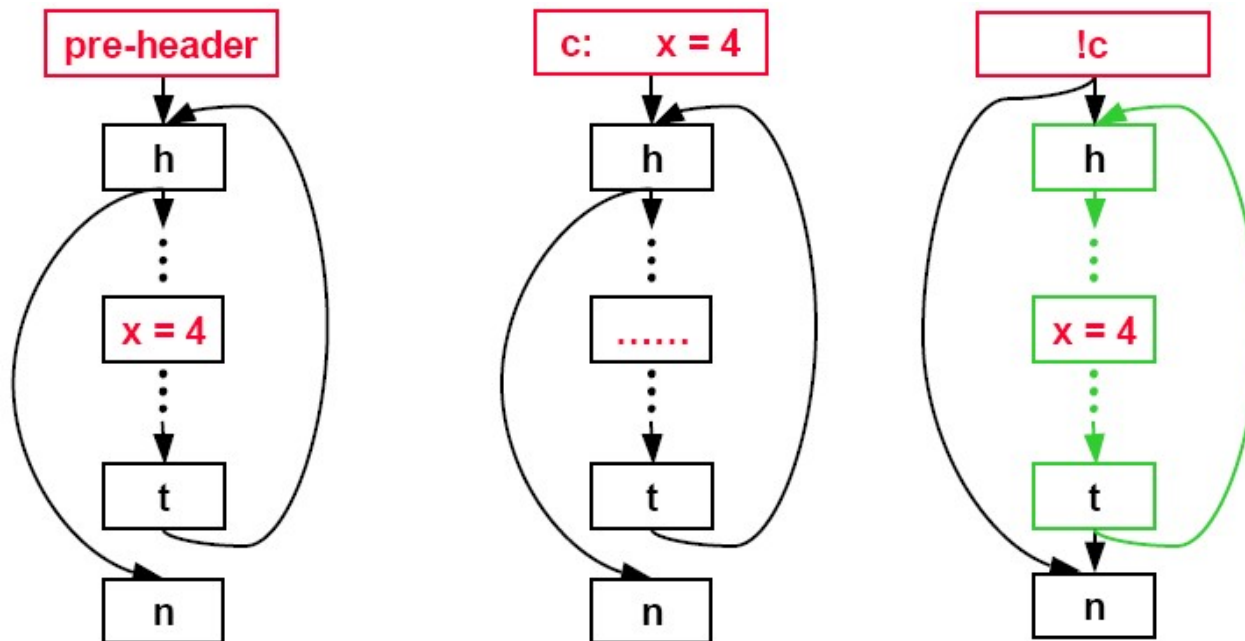
将s外提，移到L的前置结点}

}

}

4.3 循环不变量外提

- 如果循环迭代次数为0，怎么办？
- 由先测试循环是否执行来避免



思考与讨论

- 循环不变量外提算法还可以如何优化?
 - ⊕ 条件1、2是否可以放松?
- 分析LLVM的循环不变量外提算法，和讲授算法相比，做了什么优化?

`llvm-src/llvm/lib/Transforms/Scalar/LICM.cpp`

4、循环不变量外提

例子

```
#include <math.h>
void loop(double *a, const double b, int m)
{
    int i;
    while (i++ < m)
        a[i] = sin(b);
    return;
}
```

5、死代码删除

- 死代码删除 (Dead Code Elimination, DCE) 优化删除死变量
- 死变量
 - ⊕ 如果一个变量 v 在点 p 开始的某条路径上使用, 那么变量 v 在点 p 是活跃
 - ⊕ 否则, 变量 v 在点 p 是死变量
- DCE 优化试图删除被定值但是出口不活跃的变量

5、死代码删除

死代码删除算法

change = false;

循环{

 执行活跃变量分析;

 对每一条语句s定值变量v {

 如果活跃变量out(s)不包含v {

 删除s;

 change = true;

 }

 }

}while(changed)

5、死代码删除

■ C代码dce.c

```
int foo(int x, int y) {  
    int a = x+y;  
    a = 1;  
    return a;  
}
```

无dce优化 (opt dce.ll -S -mem2reg)

```
define dso_local i32 @foo(i32 %0, i32 %1) #0 {  
    %3 = add nsw i32 %0, %1  
    ret i32 1  
}
```

dce优化 (opt dce.ll -S -mem2reg -dce)

```
define dso_local i32 @foo(i32 %0, i32 %1) #0 {  
    ret i32 1  
}
```

5、死代码删除

■ C代码dce2.c

```
int b; //global variable
int foo(int x, int y) {
    int a = x+y;
    b = a;
    return x;
}
```

dce优化 (opt dce2.ll -S -mem2reg -dce)

```
define dso_local i32 @foo(i32 %0, i32 %1) #0 {
    %3 = add nsw i32 %0, %1
    store i32 %3, i32* @b, align 4
    ret i32 %0
}
```

DCE优化不能删除对全局变量b的写，因此不能删除对a的赋值

5、死代码删除

■ 下面的C代码dce3.c?

```
int b; //global variable
int foo(int x, int y) {
    volatile int a = x+y;
    b = a;
    return x;
}
```

dce优化 (opt dce3.ll -S -mem2reg -dce)

```
define dso_local i32 @foo(i32 %0, i32 %1) #0 {
    %3 = alloca i32, align 4
    %4 = add nsw i32 %0, %1
    store volatile i32 %4, i32* %3, align 4
    %5 = load volatile i32, i32* %3, align 4
    store i32 %5, i32* @b, align 4
    ret i32 %0
}
```

DCE优化通常不删除对
volatile 变量的读写

6、激进的死代码删除

- Aggressive Dead-Code Elimination (ADCE)
- 思想：假设所有代码都是“死”代码，除非有证明表明该代码对程序最终结果有贡献：
 - ⊕ 执行IO，写存储、从函数返回，或者调用具有副作用给的函数
 - ⊕ 对其他活跃代码中使用的变量赋值
 - ⊕ 是一条条件分支，并且有其他活跃代码控制依赖于它
- 遍历所有代码，删除所有未标记为“活跃”的代码

6、激进的死代码删除

6.5 激进的死代码删除

思考

阅读代码：

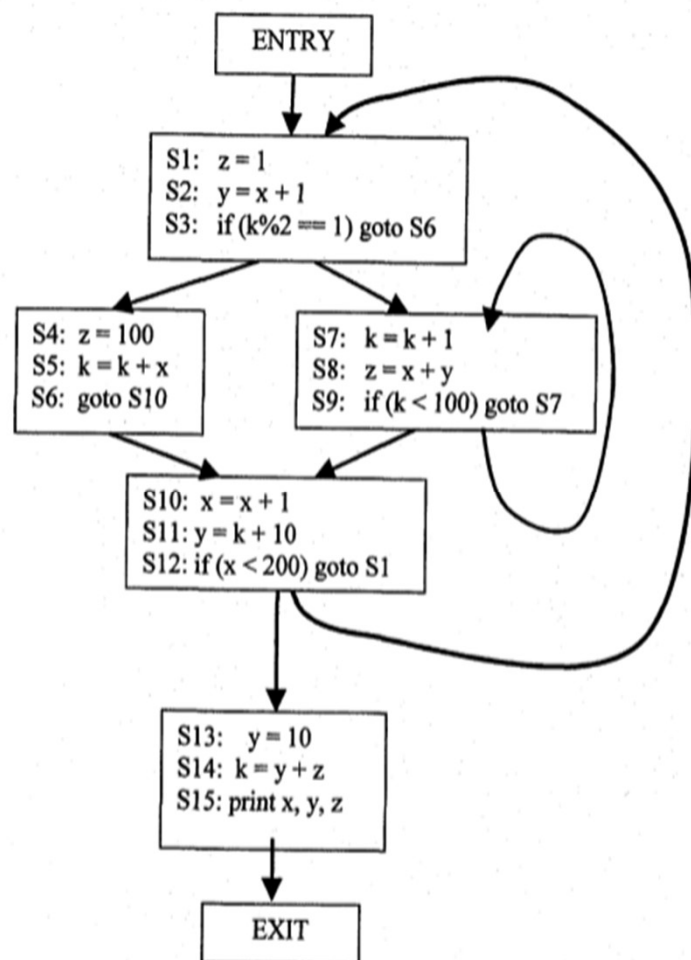
`./llvm/lib/Transforms/Scalar/ADCE.cpp`

6、标量优化

参考资料

- 教材第8章
- LLVM代码

作业



- (1) 进行活跃变量分析
- (2) 画出DCE优化后的CFG